Techniques et applications du traitement de la langue naturelle
Travail pratique 2

Classification et prétraitement de textes

Équipe 19:
Diego Silva Pires
NI: 111220136

13 novembre 2018

Introduction

Ce travail consiste à se familiariser avec des logiciels d'apprentissage automatique en proposant une solution pour faire une analyse selon la polarité “positive” ou “négative” et une autre solution pour l’identification de la langue d’un texte.

Le code pour les deux solutions peut être trouvé sur GitHub:

https://github.com/DiegoPires/IFT-7022/tree/master/TP2

Chaque solution se trouve sur un répertoire différent dans le répertoire GitHub. Les deux projets sont indépendants. Python 3.6.2 a été utilisé pour les deux solutions avec l’utilisation des bibliothèques Scikit-Learn et NLTK en utilisant Visual Studio Code comme IDE.

Parce que le projet est fortement dépendant de la bibliothèque NLTK, il est nécessaire de télécharger les quatre packages de data suivants:

```
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
```

Pour exécuter chacune des solutions, il faut simplement entrer dans le répertoire correspondant et exécuter le fichier “main.py” avec Python dans l’invite de commandes.

1. Tâche 1 - Analyse de sentiment

La première tâche proposée consiste à classifier des critiques de produits selon la polarité “positive” ou “négative”. Un corpus avec 2000 critiques a été donné pour être utilisé pour l'entraînement du modèle et pour la réalisation des tests.

Plusieurs configurations minimales pour les classificateurs ont été demandés dans ce rapport. L’objectif est de pouvoir évaluer celui avec la meilleure précision.

Classificateurs	Naive bayes, régression logistique
Normalisation	Stemming, aucune normalisation
Sélection d’attributs	Tous les attributs, avec classes, ouvertes seulement, sans les mots outils
Valeurs d’attributs	Compte de mots
Autres attributs	Nombre de mots positifs/négatifs

1.1. Solution

Deux algorithmes différents ont étés proposés pour la solution des classificateurs de sentiment. Un avec le classificateur *NaiveBayesClassifier* du package NLTK et un autre avec les classificateurs *SGDClassifier*, *MultinomialNB*, *LogisticRegression* et *LinearSVC* du package Sklearn.

Les classificateurs *LinearSVC* et *SGDClassifier* ont été choisis en plus des deux autres requis pour ce rapport parce qu'ils sont considérés de bons classificateurs pour ce type de problématique selon les documentations trouvés dans l'internet.

Aussi, avec la façon dont le code a été implémenté, c'est facile d'ajouter d'autres tests sans avoir besoin d'écrire beaucoup de code.

1.1.1. NLTK

Cette première option basée sur *NaiveBayesClassifier* a été conçue avec l'idée d'implémenter toutes les configurations possibles d'une forme manuelle, c'est-à-dire en utilisant beaucoup les concepts de *lists*, *arrays*, *slice*, etc de la langage Python et le moins possible des fonctions prêtes du package NLTK.

Une classe appelée *SentimentClassifier* a été créée pour faire abstraction du classificateur:

```
    sentiClassifier = SentimentClassifier(
       stemming=True,
       lemming=False,
       remove_stopwords=True,
       with_tagging=True)

   sentiClassifier.add_sentiment(training_set_positive, "positive")
   sentiClassifier.add_sentiment(training_set_negative, "negative")

   sentiClassifier.create_bayes_classifier(verbose=False)
```

Parce qu'il y a beaucoup de travail manuel lors qu'on appelle la méthode *create_bayes_classifier*, cette solution s'est avéré très lente. La création d'un classificateur avec les paramètres d'exemples plus haut pouvaient prendre jusqu'à 30 secondes pour son éxecution.

Les paramètres fournis ci-dessus ont eu un taux de succès de .067. Par contre, le plus intéressant est de voir le taux de succès pour les sentiments négatifs qui est beaucoup plus élevé avec 0.925. Cela indique que le classificateur est probablement mal entrainé.

```
### Naive Bayes - Results:
83/200 from positive reviews
185/200 from negative reviews
```

En essayant d'analyser la raison du fort taux de succès négatif et du faible taux positif, les critères utilisés dans le classificateur ne semblent pas problématiques:

```
Most Informative Features
                    bore = True           negati : positi =      7.3 : 1.0
                    cell = True           negati : positi =      6.3 : 1.0
                   worst = True           negati : positi =      6.1 : 1.0
                  justic = True           positi : negati =      5.7 : 1.0
                 unfortun = True          negati : positi =      5.6 : 1.0
                    poem = True           positi : negati =      5.0 : 1.0
              disappoint = True           negati : positi =      4.9 : 1.0
                    rang = True           positi : negati =      4.7 : 1.0
                   excel = True           positi : negati =      4.7 : 1.0
                  resourc = True           positi : negati =      4.7 : 1.0
```

D'autres paramètres ont étés tests lors de la création de l'objet *SentimentClassifier*. Les résultats changent légèrement, mais le modèle a toujours tendance à prédire plus de bons résultats négatifs que positifs.

C'est pour cette raison plus le fait que le modèle est très lent, que la deuxième solution a été proposée à la suite de recherches pour résoudre ces problèmes.

1.1.2. Sklearn

Des recherches ont été réalisées pour essayer d'améliorer la performance de la première solution et aussi trouver une façon de comprendre la question du modèle biaisé. La solution des classificateurs du package Sklearn semblait avantageuse.

Il est plus facile de travailler avec cette bibliothèque, principalement puisqu'il y a une grande quantité de documentations disponibles sur comment les utiliser.

En utilisant principalement la fonctionnalité de *Pipeline* et *CountVectorizer,* c'est possible d'implémenter la majorité des fonctionnalités désirées comme le compte de mots, la fréquence minimale/maximale et les mots outils.

Pour implémenter les autres fonctionnalité, une classe qui hérite de *CountVectorizer* appelé *CustomCountVectorizer* a été créée. En utilisant cette classe, il est possible de réaliser la *lemmatisation*, le *stemming* et le filtre par les mots avec les classes ouvertes.

```
    count_vectorizer = CustomCountVectorizer(
                strip_accents = 'unicode',
                stop_words = classifierTest.stop_words,
                lowercase = True,
                max_df = classifierTest.max_df,
                min_df = classifierTest.min_df,
                apply_stem = classifierTest.apply_stem,
                apply_lemm= classifierTest.apply_lemm,
                allowed_tags = classifierTest.allowed_tags,
                binary = classifierTest.binary
```

```
            )
    self.pipeline = Pipeline([
            ('vect', count_vectorizer),
            ('clf', classifierTest.classifier),
    ])
```

Malheureusement parce que la *lemmatisation* a besoin d'avoir la classification POS Tag pour chaque mot, l'implémentation est trop lente, ce qui justifie son absence dans le rapport final.

Tout cette logique de création du pipeline, implémentation des attributs et prédiction, est réalisée dans une classe appelée *SkLearnClassifier.*

Aussi, pour faciliter les tests, une classe nommée C*lassifierTestSet* a été créée. Lors de la création d'un objet de cette classe, il faut simplement déterminer les propriétés désirées pour le test et utiliser la classe *SkLearnClassifier* avec cette objet:

```
   classifiers = [
        ...
      ClassifierTestSet('MultinomialNB', MultinomialNB(), stop_words='english', min_df=0.01,
max_df=0.9, use_Tfid=False, apply_stem=False, use_open_words=False, binary=False),
      ClassifierTestSet('LogisticRegression', LogisticRegression(), stop_words='english',
min_df=0.01, max_df=0.9, use_Tfid=False, apply_stem=False, use_open_words=False,
binary=False),
      ...
   ]
   for classifier in classifiers:
       skLearnClassifier = SkLearnClassifier(data_train, data_test, target_train,
target_test)
       mean = skLearnClassifier.mean_from_classifier(classifier)
```

L'utilisation de la compte par Tf-idf étais aussi ajouté dans le pipeline, si lui est reçu comme attribut requis lors de la création du classificateur.

```
    if (classifierTest.use_Tfid):
        self.pipeline.steps.insert(1,['tfidf', TfidfTransformer(norm='l2', smooth_idf=True,
sublinear_tf=False, use_idf=True)])
```

Plusieurs combinaisons possibles ont étés testées, comme il est possible de voir dans le code source fourni dans le répertoire Github de ce rapport.

Le tableau suivant présente les précision pour les classificateurs pour le top 10 des meilleures prédictions et les 5 prédictions les plus mauvaises:

Nom du Classificateur	Mots outils	Freq min	Freq max	Tfid	Stem	Lemm	Mots ouverts	Compte	POS tag	Précision
LogisticRegression	None	1	1	False	False	False	False	False	[]	0.8025
LogisticRegression	None	1	1	False	True	False	False	False	[]	0.8025
LogisticRegression	None	1	1	True	False	False	False	False	[]	0.8
LogisticRegression	None	1	1	False	False	False	False	True	[]	0.7925
MultinomialNB	None	0.03	0.5	False	False	False	False	False	[]	0.79
LogisticRegression	None	0.03	0.5	False	False	False	False	False	[]	0.79
LogisticRegression	None	1	1	False	False	False	True	False	['a', 'v', 'n', 'r']	0.7875
MultinomialNB	english	0.01	0.9	False	False	False	True	False	['a', 'v', 'n', 'r']	0.7875
LogisticRegression	None	0.03	0.5	False	False	False	False	True	[]	0.7875
MultinomialNB	english	0.01	0.9	False	False	False	False	False	[]	0.785
LogisticRegression	english	0.3	0.5	True	False	False	True	True	['a', 'v', 'n', 'r']	0.4975
LinearSVC	english	0.3	0.6	False	False	False	False	False	[]	0.4975
SGD	english	0.3	0.6	False	False	False	False	False	[]	0.4925
MultinomialNB	english	0.3	0.5	True	False	False	True	False	['a', 'v', 'n', 'r']	0.4875
MultinomialNB	english	0.3	0.5	True	False	False	True	True	['a', 'v', 'n', 'r']	0.4875

Les résultats ont varié un peu à chaque exécution du programme, mais d'une façon générale, les résultats sont stables.

1.2. Conclusion

L'hypothèse initiale était que les classificateurs *LinearSVC* et *SGD* seraient une bonne solution pour cette problématique selon les recommandations en ligne. Contrairement à la réponse attendue, ces classificateurs utilisés avec ces paramètres de base, n'ont pas bien performé et ce avec n'importe quel attribut utilisé.

Curieusement, c'est le classificateur *LogisticRegression* sans aucun attribut fourni qui obtient la meilleure précision, c'est-à-dire un classificateur entraîné sans filtrer les mots outils,

sans éliminer des mots par fréquence minimale ou maximale, en utilisant la fréquence des mots et sans *stemming* ou *lemmatiser.*

L'utilisation des valeurs de fréquence maximale et minimale semblent être ce qui influence le plus les résultats.

Avec la suppression des mots de grande fréquence le classificateur semble gagner beaucoup en précision. Cela peut s'expliquer par le fait que les mots supprimés sont des mots plus fréquents dans la langue anglaise, sans toutefois exprimer un sentiment positif ou négatif.

Par contre, le rapport démontre qu'en utilisant les valeurs de fréquence minimale/maximales conjointement avec le filtre des mots de classes ouvertes, le bénéfice vu auparavant est annulé.

Les résultats s'améliorent lorsqu'on utilise soit le filtre par mots ouverts ou soit la fréquence maximale.

L'attribut binaire (présence de mots) n'est pas aussi performant que le comptage de mots selon les résultats.

Finalement, la lemmatisation et le filtrage par mots du type ouverts ont démontré ne pas être si importants pour les classificateurs. Leur utilisation ne diminue pas trop la précision, mais ne figure pas entre les meilleurs critères utilisés dans les classificateurs non plus.

1.3. Améliorations

Le point qui nécessite le plus de travail est que ce n'est pas toutes les combinaisons possibles d'attributs qui ont étés testées.

Les paramètres de fréquence minimale et maximale sont ceux à première vue qui ont l'air de changer beaucoup les résultats obtenus. Il serait alors avantageux de les explorer plus.

La fonctionnalité du nombre de mots positifs/négatifs n'a pas été implémentée et pourrait aussi être plus explorée comme solution.

Aussi, la lemmatisation pourrait être améliorée au niveau de la performance pour permettre son ajout dans les tests.

2. Tâche 2 - Identification de la langue d'un texte

La deuxième tâche demande la construction d'un classificateur pour identifier la langue d'un texte selon les options suivantes: français, anglais, espagnol et portugais. Un long texte dans chacune des langues a été fourni pour l'entraînement et plusieurs phrases ont été fournies pour réaliser les tests.

Il est obligatoire pour cette tâche de segmenter le texte d'entraînement en phrases et d'utiliser les modèles n-grams (tailles un, deux et trois) pour l'entraînement et la prédiction des classificateurs.

2.1. Solution

La première étape réalisée pour cette solution était de transformer le texte en phrases avec le tokenizer de NLTK selon le corpus *punkt* de la langue correspondante:

```
text = read_file('/corpus_entrainement/{}-training.txt'.format(corpus_language))
tokenizer = nltk.data.load('tokenizers/punkt/{}.pickle'.format(corpus_language))
sentences = tokenizer.tokenize(text.strip())
```

Ensuit, la classe *CountVectorizer* du package *SkLearn*, qui fournit la possibilité d'utiliser les n-grams par lettre du texte, est utilisée ainsi que les *Pipeline*s pour créer le classificateur.

```
self.vectorizer = CountVectorizer(
       lowercase=True,
       analyzer='char',
       ngram_range=(1, classifier_test.ngram))

 np_texts = self.vectorizer.fit_transform(np_texts)

 self.pipeline = Pipeline([
       ('vectorizer', self.vectorizer),
       ('classifier', classifier_test.classifier) ])

 self.pipeline.fit(texts, np_language)
```

Pour tester, une liste de cas possibles est créée. Chaque classificateur est entraîné et testé selon les textes à prédire.

```
 test_cases = [
    ClassifierTest('Naive Bayes', MultinomialNB(), ngram=1),
    ClassifierTest('Linear Regression', LogisticRegression(), ngram=1),
    ...
]

for test_case in test_cases:
    classifier = Classifier(test_case, language, sentences, verbose=False)
    for test in tests:
        prediction = classifier.predict(test)
```

Un paramètre peut-être envoyé lors de la création du classificateur pour avoir la liste de features utilisés lors de l'entraînement. Il est actuellement mis à *False* dans le code pour mieux analyser les résultats .

Le façon que le code a été créé permet facilement la réalisation de plusieurs tests avec des classificateurs différents sans écrire beaucoup de code. Cela justifie le choix de tester plusieurs classificateurs.

Puisque chaque classificateur peut être configuré de plusieurs façons, ce rapport comporte seulement les paramètres de base trouvés dans la documentation en-ligne du respectif classificateur.

2.2. Conclusion

Les tests ont étés faits avec les classificateurs: *LogisticRegression, MultinomialNB, KNeighborsClassifier, LinearSVC, SVC, DecisionTreeClassifier. SGDClassifier, GaussianNB* et *SVC* avec des n-grammes 1 à 3. En analysant les résultats on arrive au tableau ci-dessous:

Classificateur	**N-gram**	**Réussite**
MultinomialNB	1	0.55
MultinomialNB	2, 3	1
LogisticRegression	1	0.75
LogisticRegression	2, 3	1
SGDClassifier	1	0.65
SGDClassifier	2, 3	1
LinearSVC	1	0.675
LinearSVC	2, 3	1
SVC	1	0.975
SVC	2, 3	1
KNeighborsClassifier 3 neighbors	1	0.4
KNeighborsClassifier 3 neighbors	2	0.525
KNeighborsClassifier 3 neighbors	3	0.55
KNeighborsClassifier 5 neighbors	1	0.45
KNeighborsClassifier 5 neighbors	2	0.625
KNeighborsClassifier 5 neighbors	3	0.625
SVC gamma auto	1	0.3
SVC gamma auto	2	0.55
SVC gamma auto	3	0.65
SVC gamma 2	1, 2, 3	0.25
DecisionTreeClassifier	1	0.25
DecisionTreeClassifier	2	0.775

DecisionTreeClassifier	3	0.725

On note que les classificateurs *LogisticRegression, MultinomialNB, LinearSVC, SVC* et *SGDClassifier* sont capables d'avoir des prédictions avec 100% de succès dans tous les cas des tests lorsqu'on les entraîne avec plus que 2 grammes.

Parmi les classificateurs, c'est le *SVC* qui est capable de mieux prédire simplement en utilisant des modèles uni-grammes.

Ce classificateur a été entraîné avec un paramètre "linear", ce qui à première vue semble être la même chose que le classificateur *LinearSVC* fait. Par contre, la documentation *SkLearn* nous explique qu'en réalité, l'algorithme implémenté pour les deux est différent, ce qui justifie la différence de résultats.

Les autres classificateurs, par contre, ne performent pas bien.

Une analyse plus profonde de chaque prédiction montre que pour le cas du classificateur *SVC* avec un paramètre *gamma* égal à 2, la prédiction est toujours "anglais".

Dans les autres cas, c'est plutôt une confusion entre les langues romanes (portugais, français et espagnol) qui justifie les mauvaises prédictions remarquées peu importe le niveau de n-gramme ou les paramètres fournis.

Le classificateur *GaussianNB* n'a pas été ajouté dans le rapport parce qu'il retourne une erreur lors des testes.

2.3. Améliorations

Cette analyse pourra être améliorée en réalisant plus de tests sur les classificateurs qui n'arrivent jamais à bien classifier. Il faut probablement changer les paramètres.

Sinon, il faut trouver la raison pourquoi qu'ils ne performent pas bien pour ce type de problème en particulier.